User JDL - DRAFT

Version 0.4

Gabriele Carcassi, Michael Haddox-Schatz, Andy Kowalski, Jerome Lauret

September 2003

1 Introduction

This work aims to define a language for job submission in which a user describes the task he wants to be carried out in the closest way possible to his requirements. It is basically an attempt to model user requests.

In our vision, the user won't specify a list of jobs, with the details about machines and resources to be used, but rather a high level request for a task to be performed. The submission system would make a plan on how to use the middleware to satisfy the request, and translate it into jobs and resources requirements. Therefore, there is no one to one correspondence between a request and a job: a single request might result in thousands of jobs, or no jobs at all. The term Job Description Language is inappropriate, and we propose to call this specification the Request Description Language or User Request Description Language, to stress that this specification deals with user requests, and not jobs running on the grid or on a batch system.

A user request will likely be also in term of experiment specific elements, such as detector components, software components and metadata. The URDL will therefore need to be extensible to allow different experiments to interface with their specific systems (file catalogs, databases, applications). From this, derives also the need for a submission system that will allow communication with experiment specific services that will translate experiment specific information in experiment independent requests.

Another main constraint of the URDL is to allow the same request to be satisfied with different architectures, as we will specify in the use cases. Different groups might need to do things in different ways. Or the same group might need to do things in different ways in different times.

We recognize that this work will be useful only if it is done in conjunction with a modular implementation that allows different groups to finalize their experiment specific parts, and integrate their software platforms. In that case, different experiment will be able to not only share the specifications, but also tools to achieve the request translation, which will be similar but not always exactly the same.

This work in part derives from the experience on the STAR scheduler project, which had similar aims but limited for the STAR experiment. The system has been first deployed on August 2002, and was very well received by the STAR user base. The specification started as a common project between STAR and JLab. Some concepts and idea were inspired by DIAL, with which we also hope to achieve a common specification. We hope that other groups will join the discussion, since the value of a specification is in a wide acceptance.

1.1 Document outline

The document consists into 3 parts:

- Use cases, where we collect a series of functionalities that the specification needs to contain or different contexts in which the specification needs to hold.

- RDL framework, in which we present the basic concepts and mechanism of the specification. It defines the basic elements of a request, and how to extend the specification.
- Common elements, in which we use the extension mechanism presented in the previous section to define request components that are common to different HENP experiments.

2 Use cases

Here we collect a list of features we might want to have. While not all of them are feasible in a short term implementation, the specification should be able to hold for those future changes.

2.1 Different datasets

Currently, different experiments specify the dataset for a job in different ways. Among these:

- File lists: a series of physical files stored on media, typically distributed file systems (NFS, AFS, …) or mass storage systems (HPSS, …)
- Logical file lists: a series of logical names that a file catalog is able to map to a physical file.
- Queries to file catalog: a series of conditions, typically experiment's metadata, that a file catalog is able to translate in a list of physical or logical files
- Queries to databases: a series of conditions which can be used to retrieve the data directly from the database
- Named datasets: a name that can be translated, by a catalog or other mean, to a series of files or database entries

A suitable URDL must allow the description of the dataset in terms of the type of dataset definition allowed by the experiment, and only by that. That is, if the software architect of an experiment designed his system to use only one or two of these paradigms, the user should only use those.
The URDL must allow for different kind of datasets, and the system reading the request must be able to enforce type choice.

2.2 Different data placement strategy

There are different strategies used to place the data, that is to make it available for the job. Typical examples are:

- Already placed: the data is placed on well-known disk resources either by the user or by the administrator. The job will expect to find the data in a specified location.
- Placed before the job execution: the job will wait for another job which places the data, typically in a cache directory
- Placed when required: when the job tries to open the file, the system will trigger a data placement if the file is not already there
- Form database: the data will be taken from a database, which is installed and configured by the administrator

The URDL shouldn't make any assumption in how the data is placed since different experiments are doing things in slightly different ways, and a single experiment might

want to change its framework. Request will be translated to the correct scheme by the request submission system.

2.3 Different architectures

The target architecture of a request can be:

- Cluster: a group of computer managed by a batch system
- Condor flock: a group of computer managed by a batch system usually more heterogeneous than a cluster, and usually not sharing a network file system
- Super computer: a large parallel machine
- Single machine: some experiment might allow the software and the data to be installed on a standalone machine. One should still be able to use the same URDL.

An experiment should be able to either custom tailor the URDL to fit all the architectures, or to specify extra requirements in case an application needs them to run on a particular architecture.

2.4 Different dataset splitting methods

There are different schemes to split data into smaller sections and assign them to a computing resources.

- Static job splitting: in this case the entire dataset is split into manageable smaller dataset, one assigned to each job, and all the jobs are submitted to the batch system. Fail recovery means to detect which jobs failed and restart them if possible.
- Dynamic job splitting: all or just part of the dataset is split, and jobs are submitted. Depending on the results of those jobs, the rest of the dataset is split, and other jobs are sent. Fail recovery can include reassembling the dataset of the failed jobs with the non assigned dataset before resplitting.
- Consumer/producer: a master program, called producer, assigned a small portion of the dataset to different slave programs, the consumers. Every time a consumer finishes analyzing a dataset, goes back to the producer to ask for another. Fail recovery is handled by the master, that knows at each moment which part of the dataset have been processed and which haven't.

The URDL shouldn't assume any specific framework, but should allow to specify extra tag in case an application is highly dependent on the model on which is implemented.

2.5 Automatic application deployment

We want the submission system to be able to understand whether a request was made for an application that is not setup at a particular site. The submission system should

- Reject requests of application that are not available/allowed at one site
- Install the application software and its dependencies in case is possible and is reasonable, in which case there are two subcases:
    - Permanent setup: the application is installed also for future executions
    - One time setup: the application is installed and then removed when the request is satisfied

No information about the software installation should be present in the request, though. The request must have only the information needed by the submission system to

determine which application to use and what service to contact to be able to install it. Other middleware should provide the necessary information for the application setup.

2.6 Different application setup

It can be reasonable to assume that one might want to configure the same application in different way at different clusters or at different sites. Reasons can be:

- Different optimizations for batch jobs vs interactive jobs
- Different dataset splitting methods: ex static job splitting vs producer consumer.
- Different site requirement for software

One could have these setups permanently, as necessities or as choices for different circumstances, or simply in a transition phase. For example, if one wants to move from static job splitting to producer consumer, should be able to do it gradually, with minimal impact on users.

2.7 Allow implementation with non grid components

Especially at the beginning, in which all the piece of the grid are not fully operational, each experiment or site must be able to incorporate ad-hoc intermediate solutions. Examples are:

- File catalogs: many experiment have their own
- Data placement utilities
- Batch systems (directly instead of through Globus)

3 Request Description Language (RDL) framework

3.1 Overview

Given the broad requirements, the URDL can't and doesn't want to address all the specific situations and provide tags and options for all the cases: it would be impossible and it would generate a gigantic specification that no one could implement completely.
The strategy is to identify what is common to all the situations, define only that, and provide a framework for each experiment or site to define their application tags and options.
For those elements which are used across experiments (i.e. shell scripts, root) we will propose a standard more detailed request definition. Applications that are already standardized and experiment/site independent, makes it possible to achieve a request definition that is experiment/site independent. For others, such as the monte carlo simulation or the data production of the different experiments, is more unlikely.

3.2 Basic concepts

While the setup can be quite different as described in the use cases, there are always four concepts that remain the same and that constitute a request. The first three describe the request itself, while the last gives the submission system hints on how to satisfy it. They are:

- Application: a user will need to say which program is going to run for his request. The application is considered not written by the user, and is not changed at a request by request basis. It's the part of software that is managed, has release

numbers and is typically already resident at the target site. If it is not resident, its installation has to be supervised, or at least authorized, by the site/cluster administrator.

- Task: a user will need to say what to do with the program. In some cases specifying the application will be sufficient, but in most cases it won't. For example, if I specify root as an application, I will need to specify my macro; if I use csh, I will need to give a script. The task is the user defined part of the program; it can change at a request by request basis. It will be typically need to be made available from the submission site to the target site in some way.
- Dataset: a user will need to say on what data the program should run.

By themselves, Application, Task and Dataset constitute all the information needed to define the provenance of the result. That is, to define how the result was constructed. This distinction will be useful for integration with other work (i.e. Griphyn/Chimera). This approach was inspired by David Adam's work on DIAL.

There is some other information that might be needed to aid the submission system on how to satisfy the request. All this information is put in another section.

- Extra: a user might want to specify some extra information to aid the submission system. This might include resource estimators (for example, how much memory or disk spaced is used depending on the size of the input), user information that is not in the certificate, response time.

3.2.1 Open issues

Some people feel a tag describing the output. We could add an extra section called "result" for that purpouse.

3.3 XML

We chose XML as the basis for the description for the following reason:

- It fits with the current trend in the grid community toward XML and web services
- It allows us to have a hierarchical structure, in which concepts can be refined as we go deeper in the tree. This maps with the need to have very general concepts of application/task/dataset/extra that can be first refined for commonalities and then, if needed, refined in slightly different ways by each experiment.
- There are a number of tools and libraries to help us edit, parse and generate XML.
- It's being used already in STAR user JDL with success, providing a proof of concept.

3.4 Request specification

First, let's give an example of a full request:

```
<request>

  <application name="root" />

  <rootTask>
    <macro>myMacro.C(10.0, true)</macro>
  </rootTask>
```

```
  <catalogDataset>
    <query>production=2g,type=muDST</query>
  </catalogDataset>

  <extra>
    <role>MyAnalysisGroup</role>
  </extra>

</request>
```

One should notice the four concepts presented before. The request is composed by 4 parts, the application, the task, the dataset and the extras. The schema would be:

```
<xsd:element name="request">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="application"/>
      <xsd:element ref="task"/>
      <xsd:element ref="dataset"/>
      <xsd:element ref="extra"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

The idea is that the internal syntax of a task can be task-type specific, as the syntax of the dataset can be dataset specific. For example a “scriptTask” task will contain a brief script, while a “logicalDaset” will contain a series of logical file names. Experiments that will have similar requirements will hopefully merge to a joint specification of datasets and tasks.
The different type of application, task and dataset will be creating by extending the base types. This part of the specification aims only at defining the basic elements. In the next section we will describe how to extend the basic elements, and provide specification for those elements that can be common to different HENP experiments.

3.4.1 Open issues

3.5 Basic application specification

3.5.1 Description

The application represents some software installed and configured at a remote site that is able to execute a user defined task. An application will be able to execute only some type of tasks. The definition of which task is up to the application.
The application basically represents a consistent environment around the task. It could be mapped to different executables at different sites: the important thing is that they execute the same tasks with the same results.
An application can be a standard piece of software (such as root, bash/csh/tsch, java), an experiment specific one (such as root4star) or different configurations of another application.
In this document we only limit ourselves to the specification of the application inside a request, but an application in our minds has to define a lot more than that. It has to define:

- Software resources: which software packages have to be installed.
- Task verification: whether the task description is complete and valid for the application.
- Task deployment: how all the files needed by the task to run will be made available at the (possible) remote resource. For example, it uses AFS, file transfer, a CVS checkout, …
- Task installation: how the task will be installed at the (possible) remote computing resource. Does it need to be compiled, initialized, …
- Data retrival: how the task will access the data. It can be already present at the remote site, copied before the execution, copied when the job requests it, …
- Dataset splitting method: how the entire dataset can be split for the different jobs.
- Job interface: how a particular job is going to receive the decisions of the submission system. For example, how it's going to access its specific sub-dataset.

In some cases, a choice in one of this area will require the user to specify more options. These options will reside in the task specification. That is, to be able to execute the task, the application will need to know some information to copy the task (the bundle) or install it (on which OS can it compile). The application will then support only tasks that will have that required section.
To make thing more clear, let's make two example.
A csh application may define that: csh has to be installed, that a task is a script, that to run the task will compose a command line "/bin/csh script", that to deploy the task it has to copy it, that the data is already found at the target site, that the dataset consists of files, that it can assign any number of files it wants to one job, that the job will receive the file names through environment variable.
On the other have, a root application might define that: root has to be installed, that a task is a macro containing a bundle with all the files needed to run that macro, that to install the task it has to copy it and compile it, that the macro must have a "int doEvent(Event &event)" function, that that function will be called once for each event in the dataset.
As you see, application in this context is much more than a mere installed package.

3.5.2 General scheme

```
<xsd:element name="application" type="applicationType"/>
<xsd:complexType name="applicationType">
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="ver" type="xsd:string"/>
</xsd:complexType>
```

An application is basically a name and a version number. The submission system can use that name to check whether that application is present, and dispatch it to the correct version, or return an error if the application is not present. In the future, the scheduler can also contact an application distribution and installation service to install the application if it's not present.
The application definition allows extension in the form of extended types.

3.5.3 Possible commonalities

For common applications, we can share both the syntax (i.e. the application name) and the application module within the implementation framework

3.5.4 Examples

```
<application name="root4star" ver="2.0" />
<application name="root" />
<application name="csh" />
```

3.5.5 Open issues

One might want to define namespaces to avoid collision of names between applications in different experiments. Probably this will be linked to the schema validation and the use of XML namespaces. The idea is that one would define his request in the experiment namespace.
There have been some concerns for the “application”: people would think only about the executable, and not about all the other part an application is going to define. Other suggestions include “environment”, “executor”, “shell”, “framework”.

3.5.6 Implementation framework consideration

Each application name needs to be mapped to a specific program and executable. Within the implementation framework every experiment will need a way to translate experiment specific application to a more neutral request (system JDL). This can be done through plug-ins or through web services, the details of which are beyond the scope of this document.
[TODO: add a list of requirements for the application module of a submission system.]

3.5.7 Motivation for this section

The concept of application as standalone is useful for:

- Automatic package installation. By specifying at high level which application the user is going to use, the scheduler will be more easily able to check whether the software is correctly installed and even install it.
- Decoupling of the command line. By not submitting the actual command line, we allow different site (or different farms within a site) to execute the task in a different way. This allows us to use different schemes, for example, when running interactive jobs or when running batch jobs.
- It makes it easier to divert requests to different resources on an application basis. (i.e. from today, all the reconstruction is done at that location)

3.6 Basic task specification

3.6.1 Description

The task describes what the user is going to do with the application and the data. The task description is not strictly application dependent, since different application can use the same task: different application versions or different applications that map to the same application repackaged to include different reconstruction algorithms.
Typically, the task is an xml section sent to the application module, and it's the application module that is going to interpret it. The task is what the application will run, so its pretty opaque to the submission system.

3.6.2 General scheme

```
<xsd:element name="task" />
<xsd:complexType name="taskType">
  <xsd:attribute name="type" type="xsd:string"/>
</xsd:complexType>
```

Given the premises, in general the task doesn't define much. The type of the task will be the element name, which will be changed according to the type. More about this in the next session. The type allows to check whether a given application support a given type.
The work of coming up with a common URDL is essentially finding common task elements.
For example, different experiment might share the notion of a root task, or might share, among different tasks, the notion of a bundle (i.e. files that need to be copied to the target machine). In section 4 we provide examples of this idea.

3.6.3 Possible commonalities

There can be many different common defined tasks. Ideally, all the across experiment applications can have common tasks. For example, csh can be considered the application, and a task would be a command line or a script: this would map directly to what is done through a batch system now, and it's a good place to start. Experiment specific tasks can be translated into csh tasks. Tasks for common HEP application, such as root or paw, can also be provided.
We also aim to provide a sort of inheritance for task definition to provide common sections, such as output definition, that would map to common functionalities, such as moving the output file to a storage system, or adding it to a file catalog.

3.6.4 Examples

a root macro

```
<rootTask>
  <macro>myMacro.C("arg1", 10.0)</macro>
</rootTask>
```

a java application (main class, library jars)

```
<javaTask>
  <mainclass class="MyJob" />
  <classpath>
    <jar url="file:/~/jars/myjar.jar" />
    <jar url="http://www.star.bnl.gov/~carcassi/myotherjar.jar" />
  </classpath>
</javaTask>
```

a csh script (script or command section)

```
<scriptTask>
  <script>
...
  </script>
</scriptTask>
```

```
<scriptTask>
  <script file="myscript.sh" arguments="arg1 arg2" />
</scriptTask>
```

3.6.5 Implementation consideration

The task is closely associated to the application module. That is, the application module should be able to verify a task (see if it has all the information needed), install it (e.g. move the appropriate files, compile it, and make it available on a network file system), run it (that is provide a command line to actually start the job).
The way that the task will receive the job submission decisions (such as, how the dataset was split and which part was assigned to the specific job) is also application specific. For example, say a csh job will need to know on which file it will run (FILELIST) and which JOBID is assigned to the job. This will likely be passed through environment variables. An experiment framework, instead, could pass that information directly to a framework component, behind the back of the user, which would get the data from a collection or an iterator.
Therefore, one has to keep in mind the target architectures for the implementation when developing a common syntax for a task.

3.6.6 Motivation for this section

Having the task separate from the application allows us to share its description among different application, and in the end different experiments. For example, STAR has a modified version of root, called root4star. That would qualify as a different application, but the form of the task, a root macro, might be the same.

3.7 Basic dataset specification

3.7.1 Description

The dataset is the data input for the analysis. It can be specified at a metadata level (e.g. all the events with certain physics characteristic), at a logical level (e.g. a logical file in a file catalog) or at a physical level (e.g. a physical file on a particular storage system).
The dataset definition should be connected to the work being done in the analysis working group on the definition of the dataset properties.

3.7.2 General scheme

As for the task, the dataset doesn't define much per se. We will share further syntax by defining types of datasets that are common among different experiments.

```
<xsd:element name="dataset" type="datasetType"/>
<xsd:complexType name="datasetType">
  <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>
```

3.7.3 Possible commonalities

Common types of dataset that can be shared are:

- Physical file name dataset: a dataset made of files located on a disk, either on the network or on local disk
- Logical file name dataset: a dataset made of logical file names connected to a particular file catalog
- File catalog metadata query: a dataset made of the result of a query to a file catalog. The syntax of the query would be file catalog dependent, and a module within the implementation is needed to transform the query in a list of physical/logical files. The specification of a common query language for file catalogs is beyond the scope of this work.

3.7.4 Examples

```
<catalogDataset>
  <catalog>MyExperimentCatalog</catalog>
  <query>production=2g,type=muDST</query>
</catalogDataset>

<logicalDataset>
  <catalog>MyExperimentCatalog</catalog>
  <file>run20345/file01.root</file>
  <file>run20345/file02.root</file>
  <file>run20345/file03.root</file>
  <file>run20346/file01.root</file>
  <file>run20346/file03.root</file>
  <file>run20346/file04.root</file>
</logicalDataset>

<physicalDataset>
  <file>/data01/myExp/data/run20345/file01.root</file>
  <file>/data01/myExp/data/run20345/file02.root</file>
  <file>/data01/myExp/data/run20345/file03.root</file>
  <file>/data01/myExp/data/run20346/file01.root</file>
  <file>/data01/myExp/data/run20346/file03.root</file>
  <file>/data01/myExp/data/run20346/file04.root</file>
</physicalDataset>
```

3.7.5 Open issues

Some data, such as calibration data, can be seen as part of the dataset, as a parameter of the task, or as a defined property of the application. It is unclear where to put it, and maybe the URDL should provide the flexibility for an experiment to decide in the specific case.

3.7.6 Implementation consideration

Dataset definition is intimately connected with the concept of dataset and request splitting. Request splitting is the crucial point of all the scheduling process: the dataset can potentially be divided according to the different resources used by the job (i.e. files, machines, cpu time, hard disk space for output), user characteristic (i.e. which user, what role) and site specific information (i.e. this site has a short queue, local batch system practices).

Constraint on the request splitting can be put by the type of dataset, but also by the specific task and application. Since some application will be experiment specific, an implementation need to remain flexible on how the dataset is handled, and provide a framework to insert custom dataset splitters.

3.7.7 Motivation for this section

The data input is an orthogonal section to the application and task, and different experiment might have different ways to store, access and specify the dataset.

3.8 Extra

3.8.1 Description

This section should include all the information that the submission system can use to make its decisions. While the previous three sections are enough to provide the provenance of the result, they don't include any specific information that would help the scheduler translate the request in the best way. For lack of a better word, this section is called "extra". We need a better word.
Most probably this will contain other sections, such as:

- Role – user identity will be likely taken from the authentication (i.e. grid certificates), but users might have different request that need to be processed in different ways. Along these lines, there should be a user priority for the request.
- Response time - if the job is interactive, one will need to be able to request
- Job Metrics/Requirements/Resource estimator - the job splitter will need to be able to make an estimate of the jobs resource usage. This can be done either by specifying some parameters, or a full function.
- Extra batch system attributes - the user will need some extra options to pass to the batch system. For example, a job/project name, mail notification and so on. It is not clear whether some of these actually go in the task.

3.9 Extension mechanism

As we said, the basic specification provides only fundamental concepts and nothing more. The rest of the specification builds on these to provide common specification for common application, task and datasets.*Notice that if two experiment do not share any applications, or tasks, and do not even agree on how a dataset should be defined, there is no chance that they will share a common RDL*. A common RDL, or common parts of it, can exist only if the intent of different groups is similar. We believe that such similarities exist between the different HENP experiments, some of which on all the PPDG members. The extension mechanism is based on the extension scheme in XML: it allows us to substitute an element with another element of a derived type. For example, we want to specify the scriptTask Type as a derivation of the taskType, and we want to allow the scriptTask to replace task. We add to the specification:

```
<xsd:element name="scriptTask" type="scriptTaskType"
    substitutionGroup="task"/>
<xsd:complexType name="scriptTaskType">
  <xsd:complexContent>
    <xsd:extension base="taskType">
```

```
      <xsd:sequence>
        <xsd:element name="script" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

With the same mechanism, we could specify a specialScriptTaskType that derives from the scriptTask, and so on.
[TODO: still need to investigate a way to declare different pieces in different schema definitions, basically different schema files.]

4 Common elements

In this section we go into the details of which part of a request can be standardized for broad collaboration. What we envision is that these commonalities will be defined as a per need basis, and might even evolve over time. Subsequent specification, made also by other group, should be allowed to add or extend different elements. This section also function as an example on how this work can be done.

4.1 CSH and other shells

A good starting point to illustrate how to define an application and a task is to define a csh application and a csh task. It is a good starting point for the following reason:

- It's a standard application, and it's probably useful to have a RDL ready for it
- It's something well understood
- It's close to what one specifies to a batch system

The application, therefore, is csh itself, and the task is other a script, or a file containing the script. For example, we would expect something like:

```
...

<application name="csh" />

<scriptTask>
  <script>
echo Here is the time:
time
  </script>
</scriptTask>

...
```

or

```
...

<application name="csh" />

<scriptTask>
  <script filename="myscript.csh" />
```

```
</scriptTask>

...
```

The application follows directly the general syntax, but the task does not. In fact, we need a script element that tells the script to execute, or the file that contains the script. We need a schema modification to accept these new component.

```
<xsd:element name="scriptTask" type="scriptTaskType"
    substitutionGroup="task"/>
<xsd:complexType name="scriptTaskType">
  <xsd:complexContent>
    <xsd:extension base="taskType">
      <xsd:sequence>
        <xsd:element name="script" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

This allows us to substitute a task element, which was declared as taskType, with a scriptTask element, of type scriptTaskType, which extend taskType. In this way, the parser is able to validate the XML request, and we are able to add new task, application and dataset types. In much the same way, we can extend the scriptTaskType, and we can allow to specify the parts of the request that are common, while leaving to each experiment the freedom to add their experiment specifics tags.
[TODO: still have to understand how to chain XML schema definitions]
Also notice that the same definition of a script task applies for other script interpreters: bash, tsch, perl, python… The only change needed is the name of the application
The specification, though, does not end here. The job submission system will have to make some decisions: namely, will have to get the entire dataset, and split it into multiple chunks and assign it to each job. Within the script one will need to know which particular dataset is assigned to the particular instance of the script to be able to do something with it. We need a mechanism to do that, and this mechanism is specific to the application.
Within a shell script, it comes natural to use environment variables. Let's assume that the csh application will accept datasets that consists of file. We can define the following:

- `$JOIBID` – A unique number for the job
- `$INPUTFILECOUNT` – The number of input files
- `$INPUTFILExx` – The input filename

Within the script, one can use these variables to get which data needs to be analyzed.
The application needs also to define how the inputs and outputs will be transferred, and also which inputs and outputs. As we said in the use cases, this can be done in a number of ways. The csh application will need the script itself, which is already specified, but it might also need to redirect the standard streams. For example:

```
...
<scriptTask>
  ...
  <stdin filename="myfile.in" />
  <stdout filename="myfile.out" />
  <stderr filename="myfile.err" />
```

```
</scriptTask>

...
```

The files could also be defined as URIs. To do this, we modify our schema to:

```
<xsd:element name="scriptTask" type="scriptTaskType"
substitutionGroup="task"/>
<xsd:complexType name="scriptTaskType">
  <xsd:complexContent>
    <xsd:extension base="taskType">
      <xsd:sequence>
        <xsd:element name="script" type="xsd:string"/>
        <xsd:element name="stdin" type="fileType" minOccurs="0"/>
        <xsd:element name="stdout" type="fileType" minOccurs="0"/>
        <xsd:element name="stderr" type="fileType" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="fileType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="fileName" type="xsd:anyURI" use="required"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

4.2 ROOT

Root is a common application among HENP experiments, so it makes sense to have a common specification for it. Here is an example:

```
...
<application name=”root” />

<rootTask>
  <rootMacro filename=”myMacro.C” />
  <argument>234</argument>
  <argument>myFileList</argument>
</rootTask>
...
```

or:

```
...
<application name=”root” />

<rootTask>
  <root>myMacro.C(234, “myFileList”)</root>
</rootTask>
```

4.3 Datasets

Another common area is the dataset definition. There are different kinds of dataset that can be shared. For example:

- Physical file dataset
- Logical file dataset
- Metadata query dataset

```
<catalogDataset>
  <catalog>MyExperimentCatalog</catalog>
  <query>production=2g,type=muDST</query>
</catalogDataset>

<logicalDataset>
  <catalog>MyExperimentCatalog</catalog>
  <datafiles>
     <file>run20345/file01.root</file>
     <file>run20345/file02.root</file>
     <file>run20345/file03.root</file>
     <file>run20346/file01.root</file>
     <file>run20346/file03.root</file>
     <file>run20346/file04.root</file>
  </datafiles>
</logicalDataset>

<physicalDataset>
  <datafiles>
     <file>/data01/myExp/data/run20345/file01.root</file>
     <file>/data01/myExp/data/run20345/file02.root</file>
     <file>/data01/myExp/data/run20345/file03.root</file>
     <file>/data01/myExp/data/run20346/file01.root</file>
     <file>/data01/myExp/data/run20346/file03.root</file>
     <file>/data01/myExp/data/run20346/file04.root</file>
  </datafiles>
</physicalDataset>
```

Here is an XML schema fragment for the physicalDataset

```
<xsd:element name="physicalDataset" type="physicalDatasetType"
substitutionGroup="dataset"/>

<xsd:complexType name="physicalDatasetType">
  <xsd:complexContent>
    <xsd:extension base="datasetType">
      <xsd:sequence>
        <xsd:element name="datafiles" type="filelistType"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="filelistType">
  <xsd:sequence>
    <xsd:element name="file" type="xsd:anyURI" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

First, it tells that a dataset can be described by a physicalDataset of type physicalDatasetType. Then it describes the content of a physicalDataset, which consists of a datafiles element of type filelistType, which is defined separately.
Since we defined the filelistType as a separate type, we can use the same definition in different context. For example, a logicalDataset consists of a list of files plus a catalog. The schema fragment for that would be:

```
<xsd:element name="logicalDataset" type="logicalDatasetType"
substitutionGroup="dataset"/>

<xsd:complexType name="logicalDatasetType">
  <xsd:complexContent>
    <xsd:extension base="datasetType">
      <xsd:sequence>
        <xsd:element name="catalog" type="xsd:string"/>
        <xsd:element name="datafiles" type="filelistType"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

[TODO a real specification agreed over different experiment]

4.4 Resource estimators

For the submission system to be able to divide the request into jobs effectively, it must have a sense of how resource usage is going to be affected by the split. Let's concentrate on time for the moment: common wisdom says that if jobs are too small, the overhead of the submission becomes significant, while if jobs are too long, the turnover decreases, the wait for the result increases, and resource efficiency can decrease (a long job that crashes is more likely to waste resources.
One of the element that the submission system would found useful, is a time estimator as a function of some parameters decided by the split. While doing this precisely is almost impossible, in many cases a simple linear estimator can be sufficient.
We can, for example, have a linear estimator in which each job will have a setup time, and take a constant amount of time for each event, or file. For example:

```
<extra>
   <timeEstimator>time = 1 min + 10 sec * nEvents</timeEstimator>
</extra>
```

or more simply:

```
<extra>
  <timeEstimator>
    <linearEstimator setupMinutes="1" eventsPerMinute="6" />
  </timeEstimator>
</extra>
```

In the same way, memory and disk space could be constructed.

Notice that this is just a suggestion for the submission system. In fact, the submission system could have already dispatched the same task, and kept a profile for the result. In that case, it could disregard the estimator provided in the request, and use past history to make a better estimation.

[TODO would a general expression really needed? In that case, should it handle non-invertible functions?]
[TODO what other resources?]

5 STAR elements

The STAR scheduler already comes with the concept of a request that maps to multiple jobs. The request itself, though, is not structured according the proposed framework. Here we describe a possible way to reorganize the STAR scheduler request attributes according to the proposed scheme. For a complete description of the current STAR specification, refer to the scheduler manual at:
http://www.star.bnl.gov/STAR/comp/Grid/scheduler/manual.htm.
This is constitutes the minimum required from STAR to be able to submit jobs, and can be implemented in a short time by replacing the front end of the scheduler itself.
[TODO: tag arrangement is completely open to discussion]

5.1 CSH application and task

In STAR there are basically two sets of application that we will be addressing: CSH and Root. Let's start with CSH, since the STAR scheduler users specify a small CSH script in their request.
There are a couple of assumptions we will take, which are characteristic of the STAR setup:

- At each site where the jobs will be submitted, the STAR software infrastructure is present, and accessible via PATH in the same way.
- The file:/ URLs in a request are relative to the site where the request is sent.

The assumptions go together in the sense that they are symptoms of the same cause: we don't have yet a reliable way to transfer bundles. Once suitable GRID middleware is in place, we can review the file specification and add the notion of a bundle to the request.

A request with all the parameters will look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<request>
  <application name=”csh” />

  <scriptTask name=”” fileListSyntax=”paths” minFilesPerJob=””
maxFilesPerJob=””>
    <script>
...
```

```
    </script>
    <stdin URL="file:/star/u/carcassi/scheduler/out/$JOBID.in" />
    <stdout URL="file:/star/u/carcassi/scheduler/out/$JOBID.out" />
    <stderr URL="file:/star/u/carcassi/scheduler/out/$JOBID.err" />
    <output>
      <file fromScratch="*.root"
toURL="file:/star/u/carcassi/scheduler/out/" />
    <output>
  </scriptTask>

  <starDataset>
...
  </starDataset>

  <schedParameters>
    <simulateSubmission/>
    <time setupTimeInMinutes=”10” filesPerHour=”10” />
    <scratchSpace baseSpaceInMB=”10” spacePerFileMB=”10” />
  </schedParameters>
<request>
```

The application tag refers to just the name of the application: csh. We will discuss datasets lager. The task and the schedParameters sections will contain the following:

scriptTask/@name	The name of the task to be passed to the underlying batch system
scriptTask/@fileListSyntax	The format in which the fileList will be written
scriptTask/@minFilesPerJob sciprtTask/@maxFilesPerJob	The minimum and maximum number of files that can be assigned to a single job
scriptTask/script	Contains the script with the commands to be executed. Within the script, one can use the following environment variables defined by the application: • $JOBID – The id assigned to the job by the submission system • $FILELIST – The filename of the filelist where the script can retrieve the filenames for the input files • $SCRATCH – A directory for temporary data dedicated to the job • $INPUTFILECOUNT – The number of input files • $INPUTFILExx – The filename for the xxth input file
scriptTask/stdin scriptTask/stdout scriptTask/stderr	The URL tags contain the location in which the standard streams will be redirected
scriptTask/output	This section will include the description of the output of the task
scriptTask/output/file	Describe a file output of the task. It tells its name,

	as it will appear in the scratch directory, and where they should be moved.
schedParameters/simulateSubmission	If present, the system will prepare all the scripts and jobs, but nothing will be submitted.
schedParameters/time	It gives a linear estimator for the time used by the job.

5.2 Datasets

The current scheduler interface allows the input to be a mix of files, filelists and queries. It is likely that this is won't be needed in the future, and it will be better to have 3 different separate kinds of dataset: files, filelists and catalog queries.
So, for STAR, we will need 4 kinds of datasets: one describing a list of file, one describing a filelist (which is a separate file containing a list of files), one describing a series of queries to the metadata catalog and one allowing a mix of the three, for compatibility to our current specification.

5.2.1 fileListDataset

The first two can be combined in a single entity that allows two different specifications

```
  <fileListDataset>
    <file
URL="file:/star/data15/reco/productionCentral/FullField/P02ge/2001/322/
st_physics_2322006_raw_0015.MuDst.root"/>
    <file
URL="file:/star/data15/reco/productionCentral/FullField/P02ge/2001/322/
st_physics_2322006_raw_0016.MuDst.root"/>
    <file
URL="file:/star/data15/reco/productionCentral/FullField/P02ge/2001/322/
st_physics_2322006_raw_0017.MuDst.root"/>
  </fileListDataset>

  <fileListDataset URL="
file:/star/u/user/username/filelists/mylist.list" />
```

5.2.2 catalogDataset

The catalog dataset allows retrieving a list of files based on the metadata.

```
  <catalogDataset orderBy="production" >
    <catalog
URL="catalog:star.bnl.gov?production=P02gd,filetype=daq_reco_mudst,file
type=MC_reco_MuDst" nFiles="all" />
    <catalog
URL="catalog:star.bnl.gov?production=P03gd,filetype=daq_reco_mudst,file
type=MC_reco_MuDst" nFiles="all" />
  </catalogDataset>
```

The syntax of the catalog URL is specific to the catalog, in this case the STAR catalog.

5.2.3 starDataset

The starDataset will allow a mix of the three elements.

```
  <starDataset>
    <catalog
URL="catalog:star.bnl.gov?collision=&collision;,trgsetupname=&trigger;,
filetype=MC_reco_MuDst,storage=NFS" nFiles="all" />
    <file
URL="file:/star/data15/reco/productionCentral/FullField/P02ge/2001/*/*.
MuDst.root"/>
    <filelist
URL="filelist:/star/u/user/username/filelists/mylist.list"/>
  </starDataset>
```

5.3 Root4star

The requests for Root will share the datasets and schedParameter parts, and would use a rootTask definition, and a new application name:

```
  <application name=”root4star” ver=”dev” />

  <rootTask name=”” minFilesPerJob=”” maxFilesPerJob=””>
    <macro>myMacro.C("$FILELIST", "$SCRATCH/myAnalysis_$JOBID.root",
21)</macro>
    <stdin URL="file:/star/u/carcassi/scheduler/out/$JOBID.in" />
    <stdout URL="file:/star/u/carcassi/scheduler/out/$JOBID.out" />
    <stderr URL="file:/star/u/carcassi/scheduler/out/$JOBID.err" />
    <output>
      <file fromScratch="*.root"
toURL="file:/star/u/carcassi/scheduler/out/" />
    <output>
  </rootTask>
```

The only difference is that the script element is substituted by the macro element, which includes a macro call. Also, the fileListSyntax attribute since becomes a matter of the root4star application to specify.

6 Appendix

6.1 The XML schema

This is the XML schema that includes all the schema fragments defined during the document.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!-- Basic definition of the request. Application, task, dataset, and
extra are
references: this allows them to be substituted with different types
that inherit
from the corresponding "abstract" type.-->
  <xsd:element name="request">
```

```
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="application"/>
        <xsd:element ref="task"/>
        <xsd:element ref="dataset"/>
        <xsd:element ref="extra"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

<!-- Basic application: it defines a name and a version. -->
  <xsd:element name="application" type="applicationType"/>
  <xsd:complexType name="applicationType">
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="ver" type="xsd:string"/>
  </xsd:complexType>

<!-- Basic task.
TODO: is still unclear if the type should be there and how should be
used -->
        <xsd:element name="task" type="taskType"/>
  <xsd:complexType name="taskType">
    <xsd:attribute name="type" type="xsd:string"/>
  </xsd:complexType>

<!-- Basic dataset.
TODO: is still unclear if the type should be there and how should be
used -->
  <xsd:element name="dataset" type="datasetType"/>
  <xsd:complexType name="datasetType">
    <xsd:attribute name="type" type="xsd:string"/>
  </xsd:complexType>

<!-- Basic extra.
TODO: a better name would be required for this -->
  <xsd:element name="extra" type="extraType"/>
  <xsd:complexType name="extraType">
  </xsd:complexType>

<!-- Script task. This is an example of how a task can be extended for
a script.
Using "scriptTask" instead of "task" for the third element tells the
parser to
use the scriptTaskType instead of the abstract Task. -->
  <xsd:element name="scriptTask" type="scriptTaskType"
substitutionGroup="task"/>
  <xsd:complexType name="scriptTaskType">
    <xsd:complexContent>
      <xsd:extension base="taskType">
        <xsd:sequence>
          <xsd:element name="script" type="xsd:string"/>
          <xsd:element name="stdin" type="fileType" minOccurs="0"/>
          <xsd:element name="stdout" type="fileType" minOccurs="0"/>
          <xsd:element name="stderr" type="fileType" minOccurs="0"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
```

```
  </xsd:complexType>
  <xsd:complexType name="fileType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="fileName" type="xsd:anyURI"
use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- File List type. This type can be used for both datasets that
consists of a list of files,
but also in those task that need to specify a list of files.-->
  <xsd:complexType name="filelistType">
           <xsd:sequence>
      <xsd:element name="file" type="xsd:anyURI"
maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

<!-- Physical file dataset. This dataset consists of a pure list of
physical files. -->
  <xsd:element name="physicalDataset" type="physicalDatasetType"
substitutionGroup="dataset"/>
  <xsd:complexType name="physicalDatasetType">
    <xsd:complexContent>
      <xsd:extension base="datasetType">
        <xsd:sequence>
          <xsd:element name="datafiles" type="filelistType"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

<!-- Logical file dataset. This dataset consists of a list of logical
files and a catalog
identifier that will be used to resolve the names. -->
  <xsd:element name="logicalDataset" type="logicalDatasetType"
substitutionGroup="dataset"/>
  <xsd:complexType name="logicalDatasetType">
    <xsd:complexContent>
      <xsd:extension base="datasetType">
        <xsd:sequence>
          <xsd:element name="catalog" type="xsd:string"/>
          <xsd:element name="datafiles" type="filelistType"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

</xsd:schema>
```